Cray Productivity Feature Evaluation
Abnormal Termination Processing

1 Abnormal Termination Processing

The Abnormal Termination Processing (ATP) feature manages the backtraces and core files that are generated when an executing parallel program unexpectedly terminates. ATP does this by reducing the many files generated into a manageable set and by providing a viewer to assist the developer with inspection of the backtraces.

This feature is targeted for programs that execute on large numbers of processors.

1.1 Feature Description

As HPC systems have become ever larger, the amount of information associated with a failing parallel application has grown beyond what the beleaguered application developer has the time and resources to manage. The complete picture, delivered by tens of thousands of core files (one for each process), swamps both the hardware and the user's comprehension. Yet, the single core file of the original failing process is often not sufficient to study the problem. ATP selects a manageable set of processes for which core files will be written. It selects this reduced set based on grouping processes with similar state.

ATP also produces a single, merged stack backtrace tree. This tree presents a comprehensible "picture" of what the application was doing at the time of the crash. Even with extremely large processor counts, developers can see and understand the state of their application at the time of failure.

An additional feature of ATP is its capability to hold a dying application in stasis and send e-mail to the applications developer, thus providing an opportunity for the developer to access the computer system while the application is still resident. The developer can perform a STAT analysis and attach to the failing application with a debugger.

1.2 Availability for Evaluation

ATP capabilities were first introduced in the 1.0 release of the Cray Debugging Support Tools (CDT) package, which was released in September 2009. This release includes support for a single merged backtrace tree. The coalescing of core files will be introduced in a future release.

1.3 Benefits

ATP is invoked automatically (see Restrictions, below), making usage transparent to the developer. It scalably monitors thousands of ranks in an application and presents a comprehensible picture to the user describing what the application was doing at the time that it crashed. ATP can do this with production codes that fail only rarely and yet capture the essential data to provide vital clues to the nature of the underlying problem. Furthermore, it limits the amount of required data to a tiny fraction of the worst case scenario (every single core file).

Scalability is expected to increase to 100,000s of ranks in the Cascade timeframe.

1.4 Restrictions

ATP is not yet fully implemented, and so the following restrictions apply:

- The automatic invocation of ATP when an application terminates requires an OS modification which is planned for the Danube (2010) CLE release and thus not available today. However, most of the capabilities of ATP can be experienced without automatic invocation.
- Manual recompile and relink are required today.
- Core files are not currently produced with ATP, but support for these will be added in the next release of CDT .
- When ATP is running, the application cannot be checkpointed. This will be addressed in a future release.

2 Installation and Usage

The first ATP feature, the production of merged backtraces, was delivered as part of the Cray Debugging Support Tools (CDT) 1.0 release in 2009.

A pre-release of the STAT stacktrace viewer is available. Once installed, a user can load the stat module which will set up access to the executables and man pages.

2.1 Installation

First check if ATP is already installed. On the login node of an XT system, execute the commands as shown in the example below.

```
# Check whether the atp module is available on the system
$ module avail xt-atp
xt-atp/1.0(default)

$ module load xt-atp

# view the man page
$ man intro_atp
```

If the module is not available (as indicated by no output from the first command in the example), contact your system administrator.

Loading the module will set the ATP_HOME environment variable and make the `intro_atp` man page available.

2.2 General Use

2.2.1 Build

Currently, the developer needs to modify the code to include the ATP signal handler as shown in the example below. More information is in the `intro_atp` man page.

This will not be necessary in the next release of the CLE (Danube, 2010).

```
# for C code, add the following interface and call
# to the program

     void __atpHandlerInstall();

     __atpHandlerInstall();

# link (note $ATP_HOME is defined by the xt-atp module)
$ cc –o a.out $ATP_HOME/lib/atpSigHandler.o code.c
```

```
# for Fortran code, add the following interface
# and call to the program

interface
subroutine atpHandlerInstall() bind(c,name="__atpHandlerInstall")
end subroutine atpHandlerInstall
end interface

…
call atpHandlerInstall()
```

2.2.2 Invocation

ATP is invoked when an application is launched with the `atpaprun` command. It is also possible to attach to an executing application with the `atpFrontend` command. Both the invocation and the signal handler linking will happen automatically in the next release of CLE (Danube, 2010).

```
$ atpaprun –n 4 a.out
```

2.2.3 Execution

ATP comes into play when a running application takes a fatal trap. On trapping, ATP automatically gathers stack backtraces from all of the application processes and merges them into a single backtrace tree, which is stored on disk for future analysis. The stack backtrace of the first process to trap is sent to stderr. In the next release, ATP will also record (in a file) the ranks and specific traps taken by each process.

2.2.4 Analyze

Examination of stderr's stack backtrace is the first step one would take in attempting to understand the application's failure. That, and the type of trap taken, explains where and what happened to the first process that trapped. The next step is to bring the STAT viewer up on the merged stack backtrace. This view of the data will show what the entire application was doing at the time of the trap. In the next release of ATP, it will be possible to bring up a debugger on a reduced set of core files, thereby allowing examination of the data specific to various processes.

```
$ statview atpMergedBT.dot
```

2.2.5 Rerun

Based on what is learned from the analysis it may be necessary or desirable to run the application again to gather more information. ATP will not behave differently on this second run, but one could add debug output to the application or use a debugger to study specific issues highlighted by the analysis so far.

2.3 Using the Provided Example

2.3.1 Material Location

An electronic copy of the example will be provided along with this feature description. It can also be requested via one of the contacts listed at the end of this document.

2.3.2 Resource Requirement

There are no specific resource requirements other than access to a Cray XT system, along with the CDT Release.

2.3.3 Running the Example

The sample program includes a README and a script to run the example. The basic operations are shown in the example box below.

```
# set up environment
$ module load xt-atp
$ module load stat

# build executable
$ ln –s Makefile.atp Makefile
$ make

# execute (may need to use batch)
export MPICH_ABORT_ON_ERROR=0
$ atpaprun –n 4 is.A.4.atp

# view output
$ statview atpMergedBT.dot
```

2.4 Using Your Own Application

Using the above example as a guide, try this on your own parallel application. The application must use one of the parallel programming models: MPI, Shmem, UPC or Co-Array Fortran.

If you are using the MPI or SHMEM library, it is a good idea to set either the MPICH_ABORT_ON_ERROR or SHMEM_ABORT_ON_ERROR environment variables. This will cause the library to throw a sigkill if an MPI or SHMEM consistency check fails. ATP will catch the signal and produce a backtrace.

Good example candidates would be programs that are being ported or modified (that is, more likely to encounter some errors), and that use large numbers of processors such that looking at individual stack traces is difficult to manage. In addition, it would be helpful to have access to the developer of the code or someone who is familiar enough with the code to interpret the stack trace.

To experience the advantages of using ATP, the program should be run with a large number of processes (or ranks).

3 Feedback Requested

We would like to request your feedback as part of this assessment.

3.1 Experience Running Your Own Application

- Please describe any difficulty working with your own application that was different from what you expected
- Please describe what worked well and what didn't work
- In your judgment, will this feature save you time or effort in the future?
- How would you characterize the savings (fewer iterations, less data to examine, etc)?
- What would you estimate for the savings time?

4 Contact information

Don Mason
dmm@cray.com

Margaret Cahir
n13671@cray.com